CHAPTER 48

Cross-Browser Basics with JavaScript

In This Chapter

Using assisted support

Automatic updates

The floating address bar

How the Web really works

Saving time with offline browsing

Using the active desktop

The previous chapter looked at the Document Object Model. A pertinent detail of the DOM is that we need a way to look at it and manipulate it within the context of a Web page. This is where we need JavaScript — a Web-page-based scripting language.

JavaScript: The Dynamic in DHTML

JavaScript is the language of choice for the vast majority of DHTML pages. It is supported by the two major browsers (Internet Explorer and Navigator), along with other flavors including StarOffice (`www.staroffice.com`) and Opera (`www.opera.com`). JavaScript is a relatively simple and powerful language, and is in broad enough use to make it the de facto standard for Web scripting languages.

VBScript is an extension of Visual Basic created by Microsoft as a competitor to JavaScript. However, Microsoft's efforts were not as widely accepted, because JavaScript was introduced to the Web developer world first. As a result, Microsoft has added complete support for JavaScript (calling it *Jscript*) to Internet Explorer, in addition to VBScript.

However, using JavaScript to create DHTML Web pages does have a drawback. As long as there is more than one browser, there will be more than one way of doing things. Different developers will keep up with industry standards and recommendations at different rates. The result is a mess for the lowly Web author who wants to do fun and exciting things

with a Web page, but doesn't want to limit their site to only those with the latest and greatest browser. What's a poor soul to do?

In this chapter, we'll show you the basics of JavaScript, and also include some techniques for working around the inconsistencies of the world while still delivering a combination of scripts and DOM that will work for (almost) everyone.

JavaScript is an object-oriented scripting language. With JavaScript, you can manipulate many variables and objects on your page. With JavaScript and the DOM, you can change the value of all the properties of all the objects on your page. Because the DOM requires browsers to redraw pages in response to events, JavaScript becomes far more powerful with the DOM.

It's easy to confuse Java and JavaScript—after all, they appear to be closely related. Although JavaScript bases its syntax and structure on Java, the two languages are quite independent of each other and serve completely different purposes. Java is the product of Sun Microsystems, which created it as a cross-platform, object-oriented programming language. JavaScript is a product of Netscape, which developed it to enable Web developers to add programming functionality to Web pages.

JavaScript is the most widely used scripting language on the Web. Originally developed by Netscape, JavaScript has now grown beyond the realm of anything Netscape can control and is supported natively by all the major browsers. When used with the Document Object Model (DOM), you can use JavaScript to animate, display, or hide any part of your page, validate forms, and interact in other ways with the end user.

A standardized version of JavaScript is defined by European Computer Manufacturers Association (`www.ecma.ch`), which calls their language *ECMAScript*. Netscape turned JavaScript over to ECMA in an attempt to stabilize the language and make it more widely accessible to other developers. This has not prevented Netscape or Microsoft from continuing to make their own innovations and changes outside the standards created by ECMA.

When combined with the DOM, you can do many things with JavaScript on a Web page, such as

- **Create a real-time clock with a working second hand.** A DOM-compliant browser redraws the part of the page that has changed without the need for the page to reload.
- **Create a dynamic form displaying relevant fields, based on information already provided.** For example, if a visitor answers yes to an insurance form question about whether any family members have died before age 55, then a set of questions about which relatives and how they died would appear. If the answer is a no, then the next question to appear might ask whether the visitor uses tobacco or illegal drugs. This helps to avoid such techniques as "If no, skip to question 13."

- **Reward certain screen interactions, such as answering a series of trivia questions correctly, by providing a congratulatory animation.** The JavaScript can both evaluate the results of the quiz and animate a still image (or a series of images) without reloading the page and without requiring additional actions by the visitor, such as clicking a "see results" button.
- **Sort the results of a database table based on the sort order requested by the visitor without additional server requests.** Once receiving the information from the server, the client can sort the data in useful ways utilizing JavaScript and the DOM.

This chapter is a very brief introduction to JavaScript. For the full story, plus lots of examples and expert advice, check out Danny Goodman's *JavaScript Bible* (IDG Books Worldwide, Inc.).

Even with all JavaScript can do, it has limitations. JavaScript is limited to its own sandbox within the browser. JavaScript cannot manipulate files on the client computer, including creating, writing, or deleting any system files. JavaScript also cannot execute any operations outside of the browser, including launching an installer or initiating a download.

These limitations may seem like a handicap for developers, but they help to safeguard site visitors. Right now, few Web citizens fear JavaScript because of the limitations built into it—it is not perceived as a security threat. This is unlike Java and ActiveX. Many visitors have disabled the capability for their browsers to accept any of those technologies, for fear of rogue programs. JavaScript would do well to avoid any similar security scare. If the price for this is some modest limitations, then it is probably worth the price.

Adding scripts with the SCRIPT element

Now that you have an idea what JavaScript can do, you must understand how to insert your JavaScript into your page. HTML offers the `SCRIPT` element. If you want the script to be event-driven, include the `SCRIPT` element in the `HEAD`. If you want the script to execute when the page first loads, include the script in the `BODY` element. You can have both types of scripts.

The basic syntax is the same as any other HTML:

```
<SCRIPT language="javascript">
/* script goes here */
</SCRIPT>
```

Most scripts tend to be placed directly in the Web page, but you have one other option. If your script is long or if it uses functions you want other scripts to use, you can put your script into an external text file and link to it with the `SCRIPT`

element's `src` attribute, as shown in the following code. For JavaScript scripts, the file extension is usually JS.

```
<SCRIPT language="javascript" src="/javascript/lib_date.js">
/* Perhaps a comment on why the external script is needed */
</SCRIPT>
```

Although the most popular browsers (Navigator, Internet Explorer, Opera, and StarOffice) are JavaScript-capable, other browsers still do not support it for a variety of reasons. As a responsible developer, you should hide your scripts from non-JavaScript browsers by commenting out the contents of your script. A browser ignores any tags it doesn't recognize, so the JavaScript-challenged browser will see the `<SCRIPT>` tag and ignore it, and then it will see a big, long comment (that actually contains your script) that it will ignore, and finally it will come to the `</SCRIPT>` tag and ignore that.

```
<SCRIPT language="javascript">
<!-- Hide script from incompatible browsers
...script here...
// finish hiding script -->
</SCRIPT>
```

The JavaScript-capable browser, on the other hand, won't be fazed by HTML comments. It will ignore the opening HTML comment tag by accepted language convention, and then process the rest of the contents as JavaScript. When it gets to the bottom, it sees a JavaScript comment marker (`//`), and ignores that line, which includes the closing half of the HTML comment tag.

JavaScript execution

When does JavaScript script execute? That depends on where the script is and how it's written. If a script has some effect on the initial display of the page, it should run before the page is loaded. If a script needs to be ready to run when a certain condition is met on the page, it needs to appear before the place on the page that will encounter the event. If a script needs to run in the course of loading the page, then it needs to be included in the page itself.

```
<HEAD>
<SCRIPT language="javascript">
function currentTime() {
  var timeStr = ""; //declare an empty string
now = Time();
timeStr = now.getHours() + ":";
timeStr = now.getMinutes();
return timeStr;
}
</SCRIPT>
```

```
<TITLE>My Home Page</TITLE>
</HEAD>
<BODY>
<!-- rest of document here -->
```

This snippet declares a function called `currentTime` in the head of the document, but it doesn't execute yet. But, once the page is loaded, any hyperlink, form, or other page feature that wants to use `currentTime` can, because it was declared before the page was loaded. If the script was placed at the bottom of the document, then the entire page would have to be loaded before the function was available, which could create problems if the user or page tries to invoke the function before it's ready.

JavaScript is an *interpreted* language, which means it is evaluated and executed line by line. Because the JavaScript interpreter is moving through the scripts sequentially, you need to make sure that functions and other routines are declared *before* they're needed.

The choice of when the script executes is yours. If you want the script to execute when the page is finished loading, you can place it as the last script on the page, or put it in the head of the document with a reference to it in the `<BODY>` tag.

```
<BODY onload="runScript()">
```

Your document can include as many scripts as you want or need in the head and body of the document, depending only on the patience of the end user to wait for the download.

As we delve deeper into JavaScript, you need to know a few important rules, which you may have noticed in the examples so far.

First, JavaScript is case-sensitive. This means the following three statements refer to three completely different variables:

```
Rose = 4;
rose = 5;
ROSE = 6:
```

In the preceding three lines of code, a Rose is not a rose is not a ROSE — there are three distinct variables with three distinct values. If you're not careful, this can cause a real programming headache. Be sure to follow logical, predictable naming conventions so you're not left tracking insidious spelling errors in your code.

Next, virtually all JavaScript lines need to end with a semicolon. Some of your lines of code will be more than one line. You can format your lines however you want, inserting line breaks wherever it is convenient so your code is easy to read.

Remember to put a semicolon at the end of each line; then the browser knows when the line ends.

Events, event handlers, properties, functions, methods, and DOM properties and methods use special capitalization. You've already seen this unusual capitalization system. The first letter of the first word is in lowercase and then each subsequent word has the first letter capitalized, such as `onMouseOver`, `ifValid`, and `isIndexOf`.

Variables in JavaScript are like variables in any other programming or scripting language. Variables in JavaScript are simply containers for holding data. If you want to store data temporarily, or until you can move it into a container that gets displayed on the page somewhere, you create a variable. Unlike some programming languages, you don't have to define your variables before you use them. The first time the variable name appears, the script recognizes this as a new variable and assigns a type to it (integer, number, character, and so on) based on the value you give it. The type of variable is not fixed—it can be text one moment and an integer the next, depending on its usage.

Here's the quick run-down of JavaScript syntax "gotchas":

- ✦ Variable names are case-sensitive.
- ✦ Variable names must begin with an alphabetic character or an underscore. Subsequent characters in the name can be alphabetic, numeric, or underscore (notice hyphens aren't allowed).
- ✦ Variables cannot be JavaScript keywords.
- ✦ Variables can be any length, but your fingers will get tired of typing 32-character variable names. Make names long enough to be meaningful, but short enough to type accurately.

Event handling in JavaScript

You have seen the word *events* thrown around a lot in this part of the book so far. You'll remember that an *event* is any action taken by the visitor sitting at the browser. An event can also be caused by the browser, such as when the page finishes loading. Every movement of the mouse, every click of the mouse, every keystroke can generate an event. As a developer, you must decide what kinds of actions you want to take based on events. Acting on events requires `event handlers`, which are discussed later on in this chapter.

Table 48-1 shows the major scriptable events.

Table 48-1
Scriptable Events

Event	*Trigger*
Load	This event is triggered when the page is loaded.
Unload	This event is triggered when the page is unloaded (usually when another page is called).
mouseOver	This event is triggered when the mouse goes over an object on the page.
mouseOut	This event is triggered when the mouse is no longer over an object it was formerly over.
mouseDown	This event is triggered when a visitor clicks (only the downstroke of the mouse button) on an object.
mouseUp	This event is triggered when visitors release the mouse button they have depressed. Most systems handle only the mouseUp event, rather than both mouseDown and mouseUp, or only mouseDown. If visitors start to click (triggering a mouseDown), and then move the mouse off of the object (triggering a mouseOut), and then release the button (triggering a mouseUp), then normally visitors don't want any action taken.
click	This event is triggered when visitors both click and release an object.
dblClick	This event is rarely used in Web pages because Web pages rely on single clicks, but you can capture and act on a double-click, as well.
keyPress	This event is triggered when a keyboard key is depressed and released.
keyDown	This event is triggered when a keyboard key is depressed.
keyUp	This event is triggered when a keyboard key is released.
Focus	This event is triggered only in forms, when the cursor moves to highlight a field (either by tabbing to that field, by using a mouse to place the cursor at that field, or by using an access key to bring the focus to that field).
Blur	This event is triggered only in forms when the cursor is moved away from a field that was formerly in focus.
Submit	This event is only triggered in forms when the object clicked is a `BUTTON` element with a `type` of "submit" or an `INPUT` element with a type of "submit."
Reset	This event is only triggered in forms when the object clicked is a `BUTTON` element with a `type` of "reset" or an `INPUT` element with a `type` of "reset."
Change	This event is only triggered in forms when the contents of the object in focus are changed and then the focus leaves this object. In other words, if an input field has today's date in it and the visitor changes the date and tabs to another field or clicks another field, then the change event is triggered.

As a developer, you must know what events you can detect. Then you need to think about what events you need to trap and how you want your page to behave, based on those events.

As you can see, events are happening every time visitors move their mouse across your page, even if they never click anything. Think of all the mouseOver events this might generate. However, the browser only pays attention to the ones you tell it to notice. It does this by looking for event handlers. `Event handlers` are attributes you can include in the definitions of your HTML elements.

```
<INPUT type="submit" value="Buy Now!" onSubmit="return
ifValid(this.form)">
```

In the preceding code, the submit button executes the JavaScript function `ifValid()` before submitting the form. If the `ifValid()` function validates all the data, then it returns *true* and the form is submitted using the `action` attribute of the `FORM` element. If the function returns *false*, then the function should return some useful information as to why (which you must include in the function), and the form is not submitted.

```
<INPUT type="checkbox" name="Family_History" value="yes"
onClick="determineFamilyHistory()">
```

If the check box defined in the preceding code is clicked, regardless of whether the check box is clicked to be selected or clicked to be deselected, a function called `determineFamilyHistory()` is called. This function can then reevaluate the appearance of the page or other form fields based on the new value of the checkbox.

Table 48-2 shows all the event handlers, what event they trap, and the valid elements you use them with.

Table 48-2
Event Handlers

Event Handler	*Triggered on . . .*	*Valid Elements*
OnLoad	page finished loading	BODY, FRAMESET
OnUnload	page unloading (or another page loading into the same window or frame)	BODY, FRAMESET
OnMouseOver	mouse being over an object	ALL
OnMouseOut	mouse being moved off of an object	ALL
OnMouseDown	mouse being depressed	ALL

Event Handler	*Triggered on . . .*	*Valid Elements*
OnMouseUp	mouse being released	ALL
OnClick	mouse being depressed and released	ALL
OnDblClick	mouse being clicked twice	ALL
OnKeyPress	keyboard key being depressed and released	ALL
OnKeyDown	keyboard key being depressed	ALL
OnKeyUp	keyboard key being released	ALL
OnFocus	form field being highlighted either by tabbing, by an access key, or by clicking in the field	INPUT, SELECT, BUTTON, TEXTAREA
OnBlur	form field losing the focus by tabbing out of the field, by an access key, or by clicking another form field	INPUT, SELECT, BUTTON, TEXTAREA
OnSubmit	submit button being clicked or pressed	INPUT, BUTTON
OnReset	reset button being clicked	INPUT, BUTTON
OnChange	the value of any field being changed and then that field losing focus	INPUT, SELECT, TEXTAREA

Properties, methods, and functions

Properties in JavaScript work the same as properties in the DOM. Because the DOM properties are being evaluated in JavaScript, you really can't tell them apart when working through a script. Most properties can be either viewed or changed. Some properties can only be viewed.

You access properties in JavaScript by appending the property name to the object name. For example, if you have a check box called `box1`, you might take some action based on whether it is selected. In this case, the object name is `box1` and the property name is `checked`.

```
if (box1.checked) {
take some action
}
```

Another property you will use often is the length property. It applies to any object that can be enumerated, such as form elements, strings, arrays, and so on. For example, radio buttons usually occur in sets (one name is associated with multiple buttons, only one of which can be checked at one time). If you want to check to see which button is checked, you might want to loop through all the buttons. Even if

you don't know how many buttons there are, you can loop through the list by checking each button from the 0th to the last in the list (the last in the list is the value of the object.length property). In this example, `buttonSet` is the name of the radio button set.

```
for (var i = 0; i < buttonSet.length; i++) {
  take some action
}
```

One of the properties that makes the DOM useful in manipulating the appearance of the page is the display property (a part of CSS). In the following example, the display property is set to `none` when `box1` is checked.

```
if (box1.checked) {
document.all.form[0].button[3].display == "none"
}
```

What if you want to force an event? Sometimes, you don't want to wait for a visitor to click a button, especially if only one field is in the form. Event-based methods are a way to force events or perform actions with or without user interaction.

Not all methods work with all elements, but the methods available to you are the same as the events listed previously. A method is always followed by parentheses, because a method takes an action on something. For example, `focus(box1)` brings focus to whatever object is called `box1`.

How do you call a method? From anywhere in a script or from an event handler in your element definition. One example of each follows:

```
if (box1.checked) {
focus(document.all.testform.newname)
}

<INPUT type="checkbox"
onClick="focus(document.all.testform.newname)">
```

Notice in both examples, the method simply acts on whatever is in the parentheses. Generally, methods are used only in forms, and the objects of the methods (what they act on) are also in forms. Using the focus() method is a good way to move the visitor to the correct next field for data entry.

Most of what you want to do with JavaScript, you write yourself in functions. *Functions* act on some data, so function names are also followed by parentheses. *Methods* are the freebie actions you can take in JavaScript without doing any real work. Everything else you need your scripts to do requires the sweat of your own brow.

Functions are the JavaScript workhorses. Because JavaScript is an event-driven language, most of your functions will be called either by event handlers or by other functions. In earlier examples, you've seen events call functions from within the Web page, such as the following:

```
<BODY onload="initializeForms()">
```

This calls a function named `initializeForms` which should be defined in the document's head. The definition for this function would look something like this:

```
function initializeForms() {
  //Function activity here
}
```

This function will only run when its name is invoked from within an event handler or when it's called from a script or other function. It will not run spontaneously by itself. In addition to any manipulation of the DOM or other objects inside the function, it can also return a value.

```
function initializeForms() {
  //Function does work, and sets a flag called successful
if (successful) {
         return 1;
} else {
  return 0;
}
}
```

In this case, if the form was completed as expected, it would return a 1 (which is evaluated as a logical true); otherwise it ends with a 0 (logical false). You can use this feature to ensure a function did what was expected, or you can use it to return other values in a calculator-type format.

```
function circleCircum(radius) {
  circumference = 2 * Math.PI * radius;
  return circumference;
}
```

Passing data from the page to the script

For example, you have a form with two questions, Q1 and Q2; what form fields will be used in the HTML to accept the answers to the questions posed in Q1 and Q2 is not clear. Check boxes will not work because, if the answer to both questions is "no," then neither will be checked, no visitor action will be taken, no events will be created, and the form fields for accepting the names will never appear. The two working options are radio buttons, with both Q1 and Q2 having two buttons: One each for "yes" and "no," and a select list with two options, "yes" and "no."

Either of these choices requires data be passed from the page to the script. Notice all the functions in the previous example have those parentheses following them? Well, this is where you put the data you want to pass. On the HTML definition, you put the data you want to send; on the JavaScript side, you put what you want to call the data when you receive it. This sounds simple enough, but how can you put data the visitor hasn't yet entered into the function name to send it from the page to the script? Obviously, you can't. JavaScript has a way of getting around this: it is called `this`.

This

No, this isn't one of those "Who's On First" routines. JavaScript enables you to send the contents of a form field, after it is entered, or if any action is taken on the field by the visitor, by using the reserved word `this`. What `this` sends is the value of only that field. If you need to send the contents of the entire form, say, from the onSubmit event handler to validate data before you submit it, you can say `this.form`. So, if you want to send the value of Q1 radio buttons to the script, you could use the following HTML:

```
Q1. Are the twins both boys: <BR>
<INPUT type="radio" name="bothBoys" onClick="form(this)"> Yes
<BR>
<INPUT type="radio" name="bothBoys" onClick="form(this)"> No
<BR>
```

If you want to validate an entire form when the submit button is clicked, you could use the following HTML:

```
<INPUT type="submit" value="Send Now!" onSubmit="validateAll
(this.form)">
```

This submit button would submit the names and values of all the form fields to a function of your own creation called `validateAll`.

What the script sees

Sending the value of the `bothBoys` radio button in the previous example is the same as telling the script to use the value of the object `document.form[0].bothBoys`. You can see that using this property name in every reference in this script could get cumbersome fast. Instead, when you send the value of `bothBoys` using the reserved word `this`, you can choose what you want to call the field within the script.

Being able to call the field whatever you want is important. Notice the `showBoy1()` function is called by two other functions. If `showBoy1()` acted on data, rather than just displaying (or rather unhiding) form fields, then you would have to create two different versions of the same function, one to act on each specific object. This

wouldn't be at all efficient. Instead, you can create a variable to hold the value of the object you are processing in the script until you return it to the page.

```
function bothBoys ( boys2 ) {
  if (boys2 = "yes") {
       document.all.bothGirls.display = "none";
       showBoy1 ();
       showBoy2 ();
  }
}
```

Notice in the function definition line, you are told whatever data comes into the `bothBoys` function will be called `boys2` while it is inside the function. JavaScript simply creates a container of the type of the data sent to it. In this example, the variable `boys2` is only referred to once. Even so, it is far more convenient to refer to the value of `boys2` than to refer to the entire object name.

Sending form data

When you send the entire contents of a form to a script to be validated, you will be grateful for the reserved word `this`. Here is an example of a possible `validateAll()` function for a short form.

```
function validateAll (myform) {
  validText(myform.first_name);
  validText(myform.last_name);
  validEmail(myform.email);
  validPhone(myform.phone);
}
```

Without the `this` reserved word, your script would be far more verbose and prone to error.

Testing and debugging JavaScript

As you'll see when trying some of the examples in this chapter and during your own experimentation, you can't avoid testing and debugging JavaScript. When you load a page with JavaScript, the first thing the browser does is review the script for syntax. Anything it doesn't like, it tells you about — right then and there. Then, as you try to execute the script, it finds *and reports* every error along the way.

JavaScript is interpreted, rather than compiled. This means, instead of giving you a single, long listing of errors, it tells you about errors as it finds them — one annoying pop-up box at a time. Even after you figure out what all your errors are and fix them to your satisfaction, a good idea is to try clicking fields in an order that doesn't make sense, to be sure things work as they should even when visitors do unexpected things.

Cross-Browser JavaScript

Making truly cross-browser Dynamic HTML with JavaScript begins with cross-browser HTML. If your Web page breaks on old browsers without scripting, it's certainly not going to work when you start adding in JavaScript. Solid, clean, validated HTML code is the first step in any DHTML venture.

Looking for a place to start in validating? Check out the source of the standards—the World Wide Web Consortium. They have a useful validation service at `http://validator.w3.org/`.

Having a technically perfect page ensures that the DOM for your page will be valid. This in turn means it will behave in a predictable fashion when it is evaluated and manipulated with your scripting.

However, there is another wrinkle. JavaScript was introduced with Netscape Navigator 2.0, next appearing in Internet Explorer 3.0. Each subsequent browser and implementation of JavaScript has been different in some form from competitors and predecessors. How do you implement new features without breaking old browsers? Read on.

Surveying the Browser Environment

There are two schools of thought on determining which part of a script to run in a given environment. The first is to detect the platform, browser, and version of the host, using properties of the `navigator` object. Table 48-3 summarizes the properties of the navigator object that enable you to determine information about the user's system.

Table 48-3
Key Properties of the Navigator Object for Detecting a User's Environment

Navigator Object Properties	*Determines*
appVersion	Current version of the user's browser
appName	Name of the user's browser
platform	Operating system type
plug-ins	An object that includes an array of plug-ins or ActiveX controls currently configured for the browser

Following this school of thought means keeping track of every version of every potential browser, the different features and implementations between versions, and if it has different capabilities on different platforms. The following script, when executed on a JavaScript-compatible browser, will display all of the properties of the navigator object known to the browser.

```
<SCRIPT language="javascript">
function showProperties () {
  document.write("<H1>Properties of Navigator</H1>");
  document.write("<HR>");
  var displayline;
  var obj = navigator;
  var name = "navigator";
  for (var prop in obj) {
        displayline = name + "." + prop + " = " + obj[prop];
        document.write(displayline + "<BR>");
  }
}
</SCRIPT>
```

Figures 48-1 and 48-2 show the results on Netscape Navigator 6 and StarOffice 5.1, so you can get a feel for how the results differ between browsers.

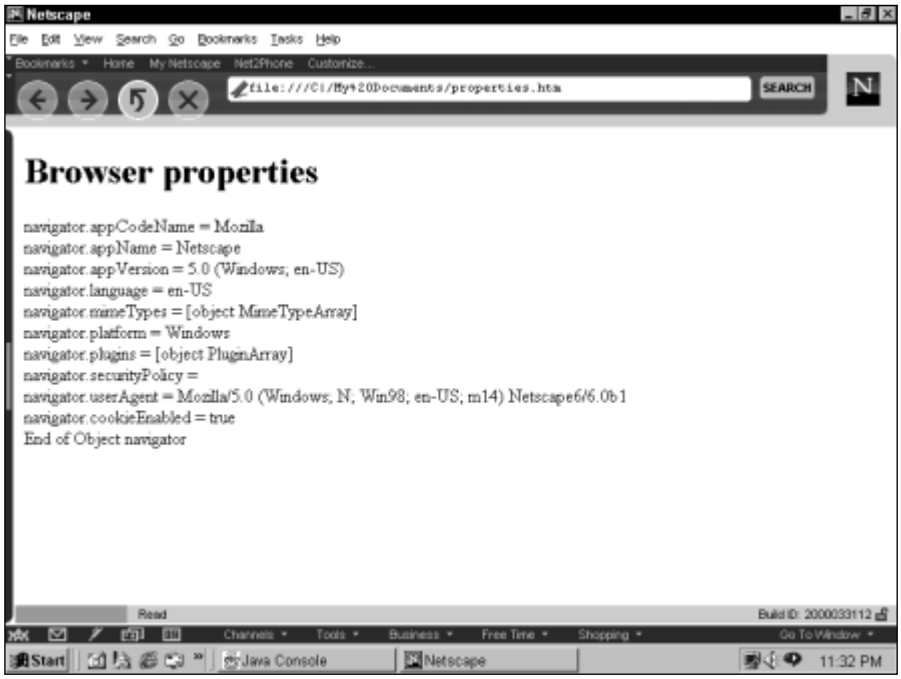

Figure 48-1: The properties of the browser and user's platform as displayed by Navigator 6

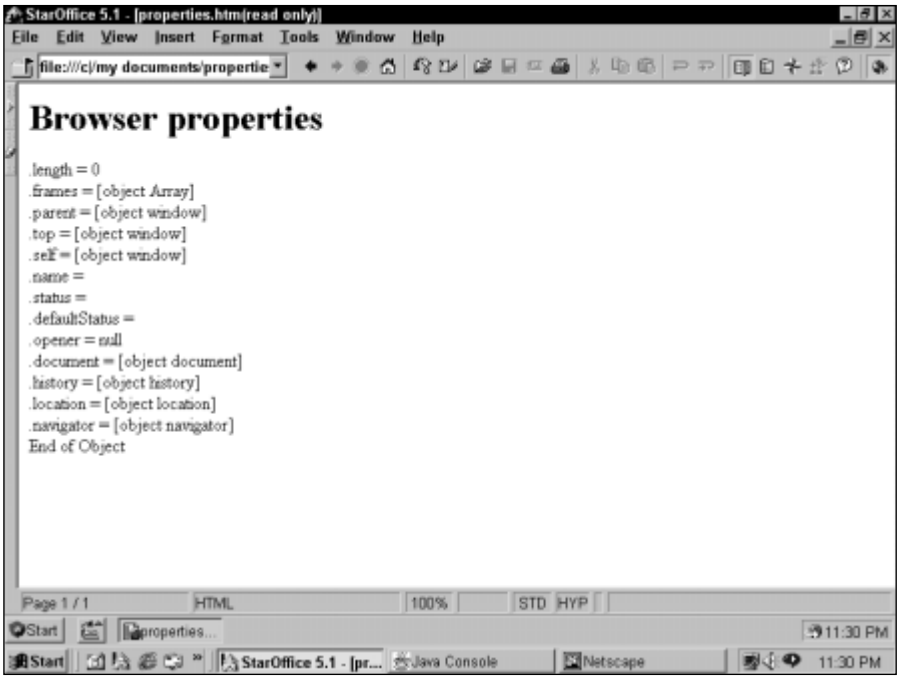

Figure 48-2: The properties of the browser and platform as exhibited by StarOffice 5.1

Cross-Reference

The people at Netscape have developed a comprehensive script for determining the browser type, version, and platform for applications that depend on specific browsers. It can be found at `http://developer.netscape.com/docs/examples/javascript/browser_type.html`. You'll still need to know the compatibilities and inconsistencies of each browser to take advantage of this. And this script doesn't allow for every instance of every browser. For example, it doesn't detect the JavaScript enabled in Sun's StarOffice application.

Sound like a nightmare? It is. Those who have remained sane in the JavaScript programming business gave this approach up a long time ago. As a sole solution, it continues to grow ever more unwieldy and unworkable given the pace of development in the browser market since 1995.

The second school of thought believes it is better to use browser detection in conjunction with object detection. Object detection can get you into the ballpark quicker, and browser detection can help fine-tune idiosyncrasies as needed.

Object detection in JavaScript

Object detection is hard to beat given the variety of browsers since Navigator 2.0, and the various legacy Document Object Models that are in existence. Object detection works by detecting which features (practically always related to an object in some form) a browser can support based on which objects JavaScript can see.

The nice feature to this method is that it provides for forward compatibility. If Netscape integrates a new object in their version of JavaScript that's not supported by Internet Explorer, then object detection will prevent the Internet Explorer user from errors generated by the unknown object. But, if Microsoft chooses to implement the new object in its next release, the feature will be enabled next time it encounters the script.

An example of object detection can be found in the following script, which determines if a browser supports the image object (not supported in early JavaScript implementations). Here's how object detection is used:

```
if (document.images) {  // if an "images" document exists
      // do something with images
}
```

If this script ran on a Navigator 2.0, the object `document.images` would be undefined and return a `0` when evaluated (interpreted as a logical *false*). This makes the `if` statement false, so it doesn't execute the statements within it.

If this script ran on Internet Explorer 4.0, `document.images` would return an object (inherently non-zero), which would be interpreted as a logical *true*. The interior of the `if` statement would run.

All of the code that depends on the image object is contained within the `if` statement that first detects if the object exists. The basic syntax of object detection is this:

```
if (name of object you want to use) {
  code which requires the object
}
```

Once you know if the object exists, you can work with it as much as you need to. This simple method will work whether you're working with math functions, images, style sheets, frames, or layers.

If Navigator suddenly starts supporting Internet Explorer's objects (or vice versa) the script will automatically "adapt" without recoding. For example, you can create a pointer to CSS layers using this method:

```
var ptrLayer;
//layerName is set to the name of the desired layer
```

```
if (document.layers) layerPointer = document.layers[layerName];
if (document.all) layerPointer = document.all[layerName].style;
```

Now you have a handle that you can use to manipulate the various CSS properties of the layer. You may have to do additional testing for specific features to ensure complete compatibility, but most of the work is done at this point. To continue to perfect the use of object detection, you'll need to know which objects you're dealing with.

When it comes to DHTML, you'll want to concentrate most on `document.layers` and `document.all`. The former essentially tells you that you're working with a Navigator DOM implementation, and the latter tells you you're working with an Internet Explorer DOM. You may want to include a test case so that if neither is true, then the browser will bypass code intended for more current browsers.

From Here

To learn more about Java Applets, go to Chapter 43.

Summary

JavaScript is the muscle in Dynamic HTML. It provides a way to examine and manipulate the DOM of its host browser, and perform other general programming functions.

As its name implies, JavaScript is a scripting language. As such, it is interpreted line by line during execution rather than being compiled into one file and executed like a traditional program. It is interpreted in the context of its environment. In this section, that means a browser.

While JavaScript has a lot of power to work within the confines of a browser, it is limited to activities within a Web page. It is not allowed to perform other actions in the user's computing environment, including accessing files, printers, or other resources on their local computer.

JavaScript is a programming language. This chapter has only given a taste of its syntax and how it works. As you work more with DHTML, you'll need to learn more about JavaScript. There are many books on the market, in addition to a plethora of online resources.

✦ ✦ ✦